

## D6.4

### Implementation: Project Knowledge Management

---

Project title:	<b>Knowledge in a Wiki</b>
Project acronym:	KIWI
Project number:	ICT-2007.4.2-211932
Project instrument:	EU FP7 Small and Medium-Scale Focused Research Project (STREP)
Project thematic priority:	Information and Communication Technologies (ICT)
Document type:	D (deliverable)
Nature of document:	P/R (prototype and report)
Dissemination level:	CO (confidential)
Document number:	ICT211932/SRFG/D6.4/D/CO/b1
Authors:	Peter Dolog, Fred Durao, Karsten Jahn, Keld Pedersen, Marek Schmidt, Rolf Sint, Stephanie Stroka
Reviewers:	Pavel Smrz
Contributing participants:	AAU, BUT, LMU, Logica, SRFG
Contributing workpackages:	WP6
Contractual delivery:	November 30, 2009
Actual delivery:	November 30, 2009

---

#### **Abstract:**

*This report presents the prototype for the project knowledge management in Logica. We describe the related theories and our approach, the 'circle of knowledge'. According to our analysis it results in the data-based import or update process between two specialized applications, one for project management and one for knowledge sharing. We describe the applications involved and the features used. The main component here is a templating mechanism in the KiWi system.*

#### **Keyword List:**

*Knowledge management, software project management, software process improvement, ontology*



# Content

<b>1. Introduction .....</b>	<b>2</b>
<b>2. Knowledge Problem in Logica .....</b>	<b>2</b>
<b>3. The Logica Circle of Knowledge.....</b>	<b>4</b>
<b>4. The Architecture of the KIWI application for Logica.....</b>	<b>6</b>
<b>5. User Guide .....</b>	<b>8</b>
5.1. Scenario Walk-Through .....	8
5.2. Application Explanations .....	12
<b>6. Logica Use Case Application Components .....</b>	<b>19</b>
6.1. Knowledge Model.....	19
6.2. Enabling Technologies .....	21
6.3. Templates .....	22
6.4. GUI .....	27
6.5. Web Services .....	28
6.6. Semantic Forms .....	30
<b>7. References .....</b>	<b>31</b>

# 1. Introduction

Knowledge management is a study and practice of representing, communicating, organizing and applying knowledge in organizations (Dolog, Krötzsch, Schaffert, & Vrandečić, 2009). In this document we will focus on an application for supporting a specific subset of knowledge management practices, namely, knowledge management for project management in organizations similar to Logica. We will focus mostly on knowledge sharing activities between and within project teams and between project teams and process design teams. As identified in (Dolog, Grolin, Jahn, Munk-Madsen, Nielsen, & Pedersen, 2008), this concerns mostly processes such as requirements management, project planning, risk management, and change management. Knowledge in such processes is constructed in evolutionary way, starting with discussions on specific issues, through classification of discussions and items until preservation in data structures for numerical analysis. KIWI system with enabling technologies can help especially in the first phases of knowledge creation and in classification and structuring of discussion and content items. Data structures used for preserving structured information which is one of the outcomes of aforementioned processes usually exist in most of the companies as part of existing IT infrastructure such as ERP systems. Therefore, we will focus on a KIWI solution for similar cases like the one briefly described above also in integration with some specific IT systems for preserving data structures. This solution provides us with an experimental set up for later evaluation of KIWI system in Logica use case carried on in WP7.

We will describe the solution in the following way. We will begin with describing knowledge management in general and the problem in Logica (Section 2). Afterwards we discuss the goal of the use case (Section 3) and a general description of the components of our prototype system and the use cases it has to fulfil (Section 4). Later, we will describe the prototype for the project knowledge management use case at Logica fitting the problems of Logica. The prototype is discussed from two perspectives: from the user's perspectives, including walk through and a handbook-like description of the three components (Section 5), and the realization perspective in with more technical overview (Section 6).

## 2. Knowledge Problem in Logica

Knowledge Management for a company like Logica is a problem on different levels. As described in the State-of-the-Art for Project Management (Nielsen & Dolog, 2008) there is a big difference between the social and the technical stand for knowledge management. For this use case we try to use a technical solution (the prototype) to solve social problems.

To do so, we start by defining our lingo, the hierarchy of knowledge: Data is raw numbers or facts, information is processed data and knowledge is possessed information by individuals. This definition is widely agreed on (Dretske, 1981; Machlup, 1980). As knowledge is something on the individual level, the distinguishing between tacit and explicit knowledge makes sense. Here explicit knowledge is expressible and tacit knowledge is not, "We can know more than we can tell", (Polanyi, 1966). Nonaka describes what to do, for transferring knowledge (Nonaka, 1994), see Figure 1.

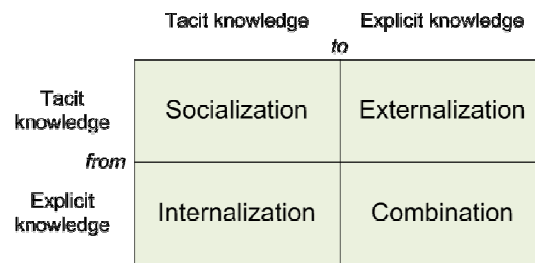


Figure 1: Modes of knowledge Creation (Nonaka, 1994)

Working on a system for knowledge management, we here mainly focus on externalization (i.e., writing down what I know) and the internalization (i.e., reading and learning, what others know). But there are alternative perspectives on knowledge, (Alavi & Leidner, 2001) defines seven of them:

1. Knowledge is personalized information.
2. Knowledge is the state of knowing and understanding.
3. Knowledge is an object to be stored and manipulated.
4. Knowledge is a process of applying expertise.
5. Knowledge is a condition of access to information.
6. Knowledge is the potential to influence action.

Each of these has different implications for the knowledge management and thus for knowledge management systems. In our case we focus mostly on two perspectives here, number 3 and number 5. The 3<sup>rd</sup> perspective's key knowledge management issue is to build and manage stocks of knowledge, where the IT takes care of gathering, storing and transferring the knowledge. In the 5<sup>th</sup> perspective knowledge management focuses on organized accessing and retrieving content and the IT's role is providing effective mechanisms for locating relevant information.

Based on our definition above, knowledge resides in the mind of individual and can thus never be threatened directly in IT. Instead we work with data or information (i.e. content items as known in the terminology of kiwi system). The role of our system is according to that and the knowledge perspectives mentioned above, to provide the possibility for users to insert, to obtain, and to find information.

In the last twenty years, various technology advancements such as Wikis (Wagner, 2004) have succeeded in support of these tasks on a collaborative level. A wiki allows users to create and edit basically all content inside, which already covers the required features from the knowledge perspectives. Most of the implementations also excel with a powerful search and users may comment on content, both increases the collaborative possibilities. Recently a new form of communication became popular, the Social Web. The idea is, to use the connections that people (or contacts) in social platforms have, to share knowledge (Dolog, Krötzsch, Schaffert, & Vrandečić, 2009). An application that realizes this again supports the perspectives of knowledge as contacts can help finding or can directly provide content of interest.

Even though these considerations are very general an analysis of Logica (Dolog, Grolin, Jahn, Munk-Madsen, Nielsen, & Pedersen, 2008) made clear, that their problems are quite similar:

- Collaboration and knowledge sharing is difficult through the different projects and employees.
- Large gap between documented and applied knowledge.
- Finding desired information is too difficult.
- Complexity between projects is too big.

- Found information has the wrong formats.

These map directly to the perspectives of knowledge described above. But Logica has an additional requirement: The existing IT infrastructure has to be respected. Logica has a certain set of highly specialized systems that organize processes, projects, employees or financial tasks. They are called Enterprise Resource Planning (ERP) Systems. Switching to different systems is usually not an option, as it is too expensive (training employees, licenses, etc) and data would be lost (complete imports are hardly possible).

Using the KiWi system (Schaffert, et al., 2009) is thus only possible under the premise that it is part of the infrastructure and does not try to replace it. While the KiWi system itself would be able to manage the knowledge according to the perspectives described above (Schaffert, et al., 2008), the Logica requires it to access data, which is already present. In other words, to follow the hierarchy of knowledge, knowledge sharing and management with the KiWi system for Logica can only work if the information inside the KiWi system is based on data from different applications that can be used independently.

### 3. The Logica Circle of Knowledge

The project knowledge management use case supports ordinary day-to-day collaboration, knowledge sharing and execution of project management tasks within and between software development projects. The project knowledge management use case application consists of two integrated parts: a sophisticated project management application (e.g., containing highly advanced planning functionality) and the KiWi system. As illustrated in Figure 2, the two applications are integrated through a shared knowledge model (a semantic model).

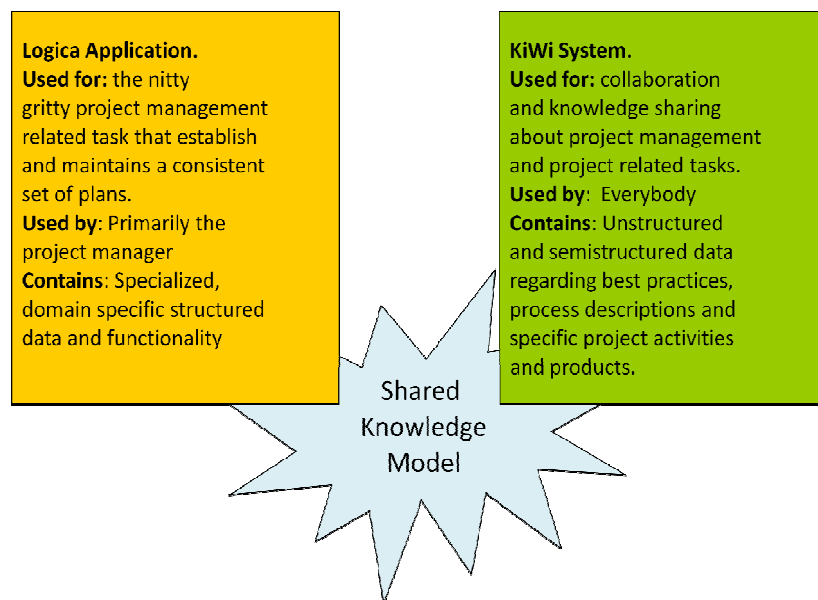
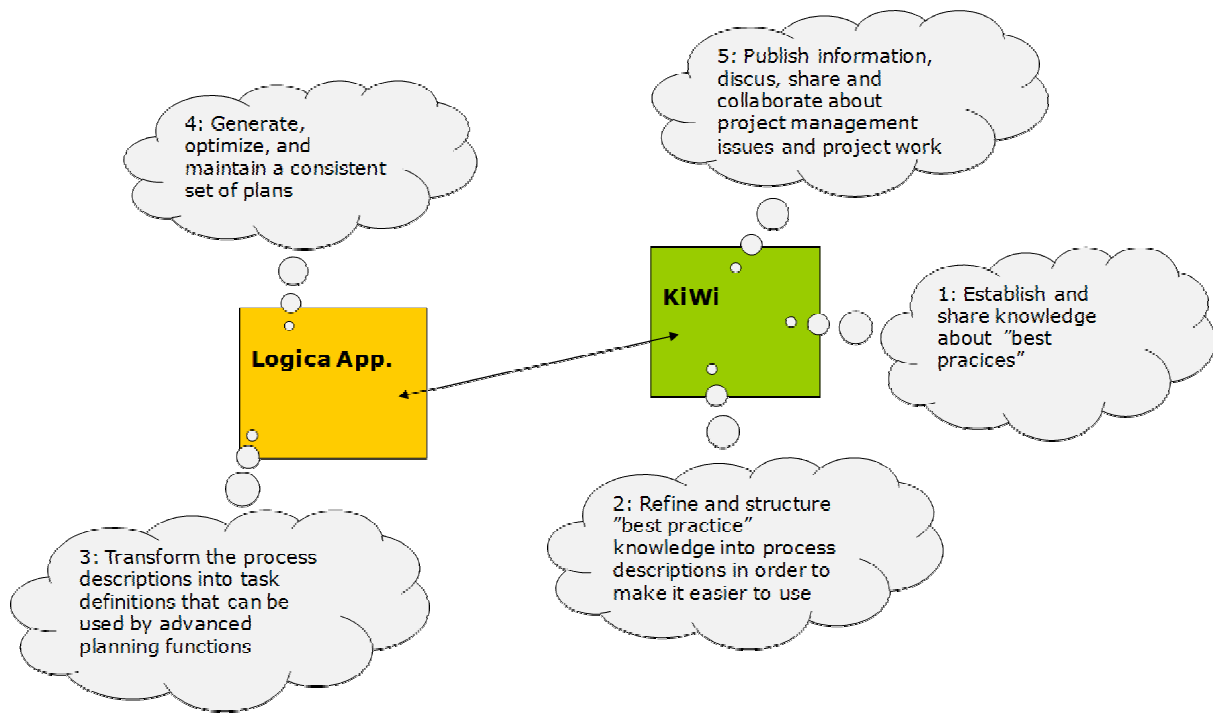


Figure 2: Connection of both applications through the shared knowledge model.

The application supports what we call the circle of knowledge through seamless integration of the two different worlds.



*Figure 3: The circle of knowledge for Logica.*

The circle of knowledge (Figure 3) consists of five activities that extract and create shared knowledge about project management (or another discipline). Our example covers process management and starts (cloud 1 in Figure 3) with classical wiki collaboration about best practices:

- Everybody can contribute "the wiki way"; adding, sharing and commenting on project management related "best practices" based on their experience, etc.
- Standard KiWi functionality used, the shared knowledge model is used for tagging.
- *Result:* A set of tagged, but un-structured wiki pages...

The next activity in the circle (cloud 2) is to refine and structure this knowledge into process descriptions:

- The unstructured wiki pages containing experience and best practice information are refined and structured into process descriptions that make the knowledge more operational and easy to use for others.
- Standard KiWi functionality is used, the shared knowledge model is used for tagging, links are inserted to "best practice pages" – the process descriptions are made using a template, and systematically tagged
- *Result:* Easy-to-use knowledge, linked to other pages containing the original un-processed information

In the following activity (cloud 3) the process descriptions are transformed into sequences of task definitions that can be used by advanced planning functions to create project plans based on established and shared best practices:

- The process descriptions in the KiWi system are used to prepare general tasks lists in the Logica application – this transformation is not yet supported by the system, but could be since the process description in the KiWi is tagged and structured in a template...

- Relevant tags from the shared knowledge model can be used to tag the task list in the Logica application and relate information in the two systems.

Afterwards (cloud 4) the advanced planning functionality is used to prepare a set of consistent plans. Based on task lists, product information, and metrics draft plans are made that are:

- Consistent, etc...
- Officially agreed on by the process manager
- Based on the "best practice" information in the KiWi

Finally (cloud 5) in the last step the plans are made public and shared by publishing them to the KiWi system for various purposes:

- *Early iterations in the circle of knowledge:* Information gathering in the early planning phase: Structured, but empty templates, are published on order to facilitate information gathering and discussion about project management related issues for example template based pages for collecting and discussing information about project risks
- *Later iterations in the circle of knowledge:* Quality assurance and commenting on "almost finished plans": Content (parts or complete) plans are published in the KiWi system for discussion, commenting, obtaining commitment, quality assurance.
- *Final iterations:* Supporting actual work by publishing "work package" information and approved plans:
  - The KiWi is the place where participants get the project related information they need.
  - The KiWi is the place where people collaborate about tasks.

The activities described below form a circle, as the output of the last step of one iteration can be the input for another. From this perspective it can also be seen as a spiral. However, in the easiest way, it can be seen as a list of activities to be followed.

## 4. The Architecture of the KIWI application for Logica

The Project Management Use Case System consists of three elements<sup>1</sup>:

1. **The KiWi System.** The main application and the location of the collaboration work. Data is inserted from an ERP System and wired to the project management ontology. Later the data can be updated or exported from here. Included information takes advantage of the enabling technologies inside the KiWi System (Schaffert, et al., 2008).
2. **The Logica Application.** This is an ERP system that Logica uses for project management. We use this as an example for any possible ERP system and do not change the code of it in any way, we access the data through a provided interface or by accessing the database directly. This application is used only by the manager or a rather small circle of people, cooperation on the data should take place in the KiWi System.
3. **The Data Exchange Application (DxA).** The tool acts as a middle-tier between the the Logica Application and the KiWi System, to organize the data integration. This works service-oriented through open interfaces of KiWi web services. It is a standalone tool that is loosely coupled to the other applications.

---

<sup>1</sup> Note that this is just a short overview. Details about the usage and implementation follow in later sections.



These three system components are loosely coupled and only make use of open interfaces. In combination an exchange of data is possible. From the usage perspective, this can be described the best through two use cases:

- Import new data to the KiWi System (UC1)
- Update already imported data (UC2)

In the update use case we also distinguish between updating the data in the Logica Application (UC2a) and in the KiWi System (UC2b).

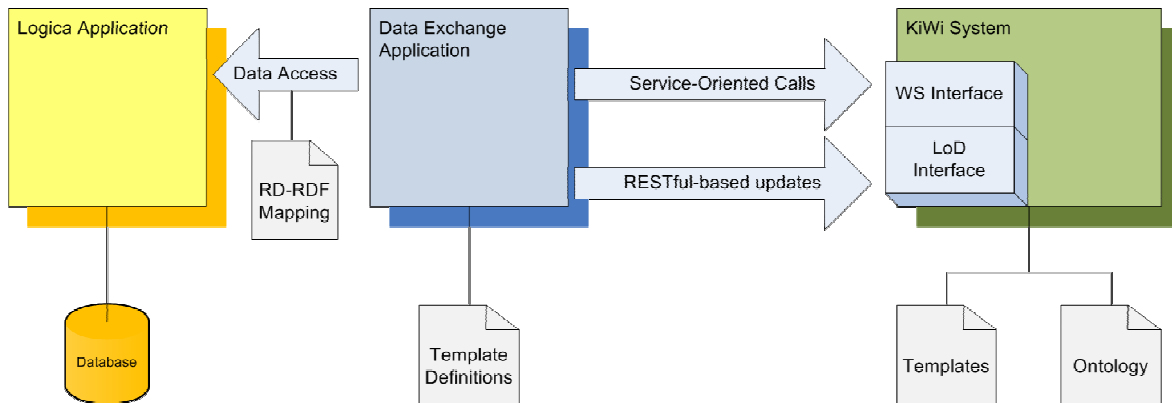


Figure 4: The System Architecture.

As described in the previous section, a shared knowledge model accomplishes the communication between both applications, through mapping an OWL ontology to a data schema in a relational database. Also web services are involved, SOAP based as well as RESTful ones that follow the LOD idea<sup>2</sup>. All ontologies and data contained in the KiWi System made accessible through certain URLs, with RESTful interfaces.

The idea of bringing data from one application to the other is realized through templates in the KiWi system. These give a frame in form of text blocks into which the import publishes data. The intention of templates<sup>3</sup> is accordingly translating raw data into an article and thus transforming it to information, which can be internalized by the users to build actual knowledge (see definitions in Section 2).

The two use cases, which we will describe in the following, are the functions that the system has to use for moving the data between those two main applications (the KiWi system and the Logica application). The five activities as described above in the circle of knowledge (Section 3) make use of this functionality, when it comes to getting data from one system to the other.

#### 4.1.1. UC1: Import

The DxA provides a table-based view on data that is needed to fulfil a template. Sorted after templates the user can thus choose the data of interest. After marking the row, the user has to click on a button “publish” to start the import process.

The DxA takes the chosen data and creates an object for the import. And it also adds the information, where the data is coming from, in form of an URI. This makes it possible, to find out later, which field on the KiWi pages were based on which in the Logica Application’s database. The

<sup>2</sup> LOD: Linked Open Data. The idea behind this is to publish data so that clients can access it, link it, etc.  
See: <http://www.linkeddata.org>

<sup>3</sup> How to use templates and what technically happens is described in later sections.

collected data (formatted in RDF and sent as a String), the data's URI and the URI of the template itself is then sent to the KiWi System, calling its web service.

The KiWi System receives the data and creates a new page, based on the template, by using the external URI, and exchanges the preset data (template placeholders) to the received data.

#### **4.1.2. UC2: Update**

As soon as data is shown inside the DxA (as described above), it accesses the KiWi System to check whether the data in the both systems are equal. This is done through the LOD interface of the KiWi System, where the DxA searches for the correlating data by the external URI. If the data in a field is different in both systems, this field is highlighted. The user has now the chance to update the information inside the Logica Application or the KiWi System by clicking a specific button.

##### **UC2a: Update the Logica Application**

The DxA contains the data from the KiWi System for comparing. To integrate that data to the Logica Application, the DxA accesses the database directly to update the specific fields. After everything is updated, the view of the DxA refreshes, so that the new data is visible.

##### **UC2a: Update the KiWi System**

The data update for the KiWi System works analogue to the import. The DxA creates an object that contains the update information and certain meta-data. It will then send this object to the same web service, which, now that the page already exists, picks the originally created one. There then the data is substituted. As after the import, a positive answer contains the URI of the updated page, so that the DxA can call it for the user immediately.

## **5. User Guide**

This section explains the systems and how they follow the approach described above. Our focus here lies in the *circle of knowledge* (Section 3), which we will explain in a concrete scenario. Afterwards we provide details about the different systems in a rather handbook style.

### **5.1. Scenario Walk-Through**

To follow the process described as the *circle of knowledge* we will now follow the process manager Paul with his work to improve the process for auditing. We will explain the steps that Paul has to follow on the applications to do so. Find a more detailed description of the system's use afterwards (Section 5.2).

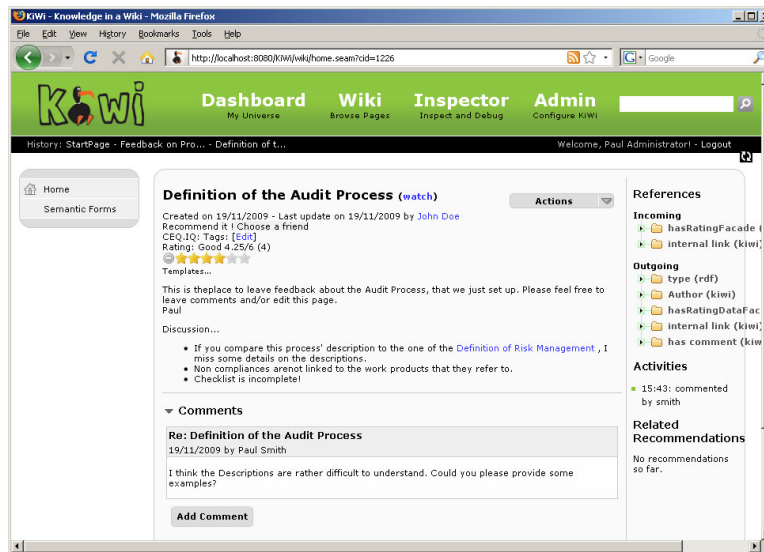


Figure 5: Knowledge sharing in the KiWi system.

### Step 1: Establish and share knowledge.

Paul creates a page in the KiWi system where he asks for comments or other feedback regarding the audit process. He then uses the KiWi search and recommendation to find people, that work with this process and invites them to give feedback as well. He then watches the page and the discussion, contributes, when needed, but mostly just reads the comments (Figure 5).

To create a page, he sets a link on another page (editing the content of any page by “Edit” from the “Actions”) to the one he has in mind (e.g. [[Definition of the Audit Process]]). Clicking on it automatically opens the edit screen for the new page. To invite others, he simply sends the link via e-mail.

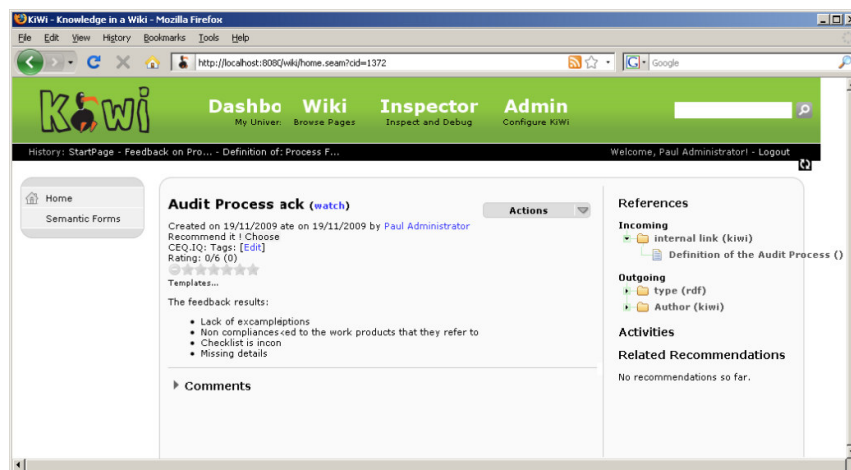


Figure 6: Refining the collaborative results.

### Step 2: Refining Discussion.

After a while, when he does not expect any further input, Paul refines the discussion's result. He might do that on a dedicated KiWi page which links to the discussion page or directly underneath on the same page. However, he analyzes the feedback and creates a list of things to change (Figure

6). Editing a KiWi page works basically like standard text processing tools do. The main functionalities are represented in icons of the edit screen. To enter the editing mode, choose “Edit” from the “Actions” menu and click on “Save” after editing to persist the changes.

The screenshot shows a web browser window with the title "smdt - Mozilla Firefox". The address bar displays "http://localhost:8080/smdt/ProcessDefinitionEdit.seam". The browser's menu bar includes "File", "Edit", "View", "History", "Bookmarks", "Tools", and "Help". Below the browser window, there is a navigation bar with tabs for "smdt", "Home", "Project", "Organization", "Process", and a "Login" button. The main content area is titled "Add process definition" and contains the following fields:

- Name \***: A text input field containing "Audit".
- Process area \***: A dropdown menu with "Measurement and Analysis (MA)" selected.
- Employee \***: A dropdown menu with "John Doe" selected.
- Check list by entry criteria check list**: A dropdown menu.
- Check list by exit criteria check list**: A dropdown menu.
- Input description**: A text area containing two bullet points:
  - Audit plan
  - Work products produced by process to be audited
- Output description**: A text area containing three bullet points:
  - Audit report
  - Non-Compliances
  - Corrective action
- Description \***: A text area containing a numbered list:
  1. Arrange audit
  2. Review work product
  3. Interview project participants
  4. Write audit report
  5. Agree on non-Compliances
  6. Propose corrective actionfollowed by "-> Examples!".

Figure 7: Inserting collaborative results to the Logica application.

### Step 3: Create new entity.

With this list he creates a first version of the new audit process definition. He does not just copy and paste the refined feedback results but inserts data bits and bullet points according to that. He understands it as a draft version and wants to develop it further at a later point in time. This version is basically a more precise version of the refined feedback results plus general input from Paul's experience as a process manager (Figure 7).

In order to do so, he opens the Logica Application in his browser and uses the main navigation to open the entity of concern, in this case *ProcessDefinition*. As he wants to enter a new data set, he clicks on “create new” on top of the page.

Paul might publish it directly to the KiWi system to get feedback on this draft. In this case, he jumps to step five, then repeats the first step and updates the data according to the comments.

**Add process definition**

Name \*

Process area \*

Employee \*

Check list by entry criteria check list

Check list by exit criteria check list

Input description  
Before starting this process an audit plan has to be created. Also the work products have to be produced by a certain process to be audited.

Output description  
The process's output is an audit report, the non-Compliances and possible corrective actions to be taken.

Description \*  
The auditor arranges to hold the audit in consultation with the project manager. With the help of the project manager or designated project participants the relevant work products are identified. The auditor reads work products in preparation for the audit.  
In accordance with the audit plan the audit interview are held and notes of agreed non-compliances are prepared. The non-compliance notices are provided with the auditor's proposed action plan. Work acceptance notice sent home on

Figure 8: Progressing the work in the Logica application.

#### Step 4: Improve entity's data.

Of course the data needed for the process definition is not just given through the feedback and Paul's general knowledge. He has to take care about consistency with the process guidelines, resource planning and other related topics. As a process manager it is his job to provide a complete process. But he starts by formulating the bullet points to complete sentences and defining the data he just put roughly before (Figure 8).

In the Logica Application he browses to the list of ProcessDefinition using the main navigation and clicks edit behind the entity created in step three. This leads him back to the editing screen where he can change the entries and clicks "Save" afterwards to persist his changes.

**KiWi Project: Data Exchange Application (DxA)**

Fetch Templates Tools

**Template List:**

- Employees Template
- Organizational Units Template
- Process Definitions Template

**Data View:**

name	input_description	description
Test-driven develop...	Lorem ipsum dolo...	Lorem ipsum dolo...
Unified Process (UP)	Lorem ipsum dolo...	Lorem ipsum dolo...
V-Model	Lorem ipsum dolo...	Lorem ipsum dolo...
Waterfall model	Lorem ipsum dolo...	Lorem ipsum dolo...
Wheel and spoke ...	Lorem ipsum dolo...	Lorem ipsum dolo...
When it's ready [1]	Lorem ipsum dolo...	Lorem ipsum dolo...
Worse is better	Lorem ipsum dolo...	Lorem ipsum dolo...
You Ain't Gonna Ne...	Lorem ipsum dolo...	Lorem ipsum dolo...
Zero One Infinity	Lorem ipsum dolo...	Lorem ipsum dolo...
Win-Win Model	Lorem ipsum dolo...	Lorem ipsum dolo...
Audit	Before starting thi...	The auditor arran...

Update KiWi System Update LUC Publish

Figure 9: Publishing the data to the KiWi system.

#### Step 5: Publish data.

As Paul considers the process to be defined and done, he publishes it. For that he has to open the DxA, chooses the template for process descriptions and then the audit process (Figure 9). By clicking the button "Publish" the data transfer to the KiWi system starts. After the successful

import, he browses to the KiWi page with the new created process definition (Figure 10). The new process is ready for application.

The publishing is handled in the DxA, it monitors the published data and checks whether published data has been changed. After starting it, Paul has to ask for the Templates by clicking on “Fetch Templates”. The DxA retrieves a list of templates and displays it. Paul then chooses the *ProcessDefinition* template, which triggers the DxA to provide a list of all data sets the Logica Application contains, which are possible for import. The DxA shows a list of all the data fields that are affected for the import. Paul chooses the row that he created in the Logica Application concerning the *Audit Process* and clicks on “Publish”. After the DxA finished the download successfully, it provides the user the possibility to open the created page directly in a browser.

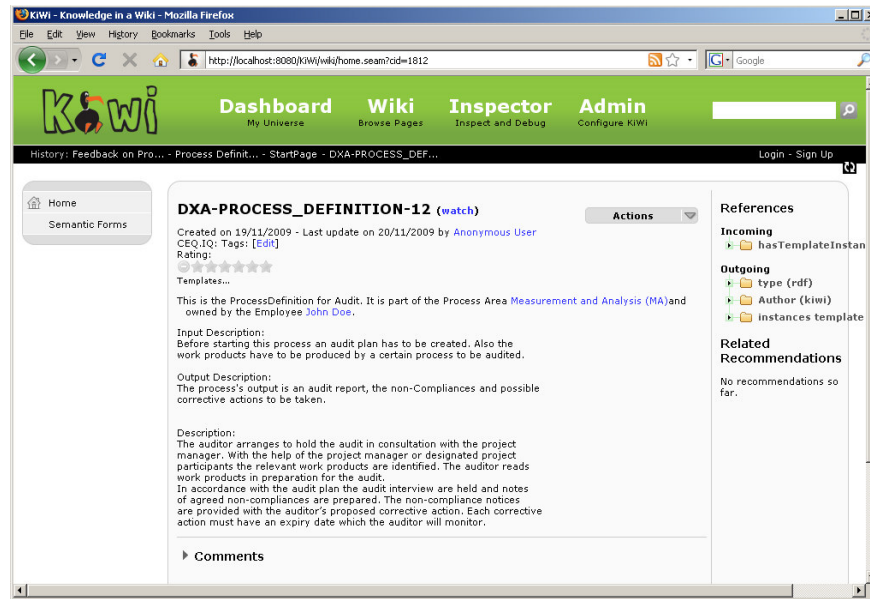


Figure 10: The imported data in the KiWi system.

## Further Steps

Paul wants to create processes of high quality. The users should not have problems to follow them or miss certain things. He also wants to make sure, that the users understand the process and the reasons for detail decisions, etc. That's why he links the process to the pages of collaboration during the creation. Feedback and further changes discussions take place on the original discussion pages or in the comments on the process page itself. He includes changes that do not require the support of the Logica application directly right on the KiWi page. Other things are changed in the Logica application and Paul updates the given pages afterwards again using the DxA.

## 5.2. Application Explanations

In the following three sub-sections we will explain the use of the applications in detail. In a handbook-style we describe the interfaces and how to handle them. We also illustrate what to do, to follow the scenario described above. However, it is not directly connected to the scenario and should be seen as a user manual.

### 5.2.1. KiWi System

The KiWi System provides templates. They define how to create new pages of specific type during the import. The creation of the templates is not different from the creation of any other page. It

starts with putting a link on a page (e.g., the StartPage) to the template page a user would like to create. Note that we will not describe the general features of the KiWi System, but focus on those that are needed for the data import.

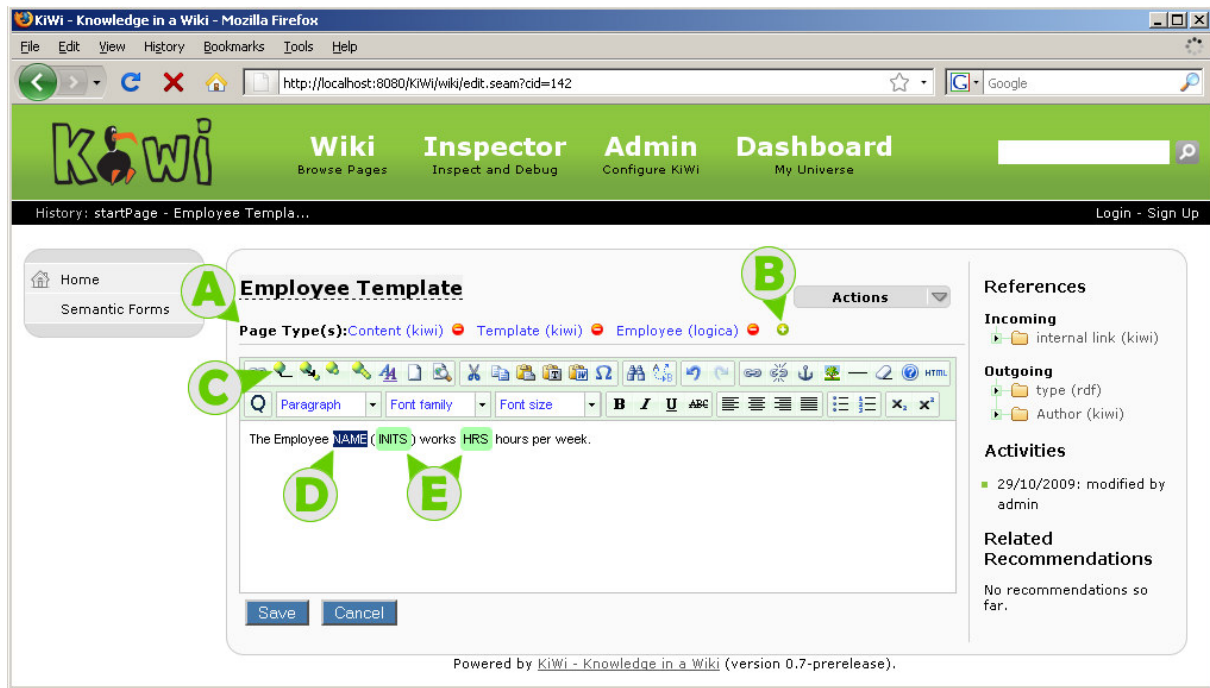


Figure 11: Creating a template in the KiWi System.

Figure 11 shows the edit screen of a page, which should become a template. We use alphabet labels to indicate what needs to be done in order to create a template. We have to add page types (A) by clicking the green “plus” button (B). As the page should be a template, we have to choose the type “Template” from the KiWi ontology. For importing data from the Logica Application, we have to choose the specific type from the project management ontology as well. The next step is to create text and connect elements from the text to the ontology, so that KiWi knows where in the text the data can be found or needs to be inserted for particular property. To do so you have to select text (D) and click the adequate RDFa button<sup>4</sup> (C). A small window pops up, which allows users to choose from the properties defined to the defined types (Figure 12). Afterwards the allocated text is highlighted (E).

<sup>4</sup> We will explain these buttons next; RDFa and technical details are described in Section 6.3, in the technical part in this document.

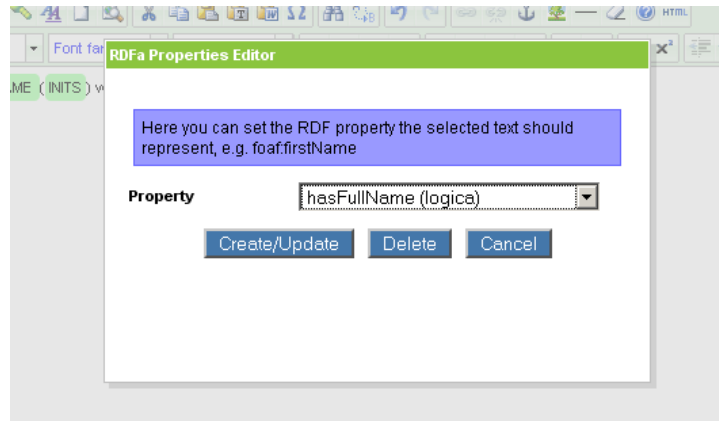


Figure 12: Choosing an RDFa property.

The RDFa buttons are the user interface for connecting the text with the types of the ontologies. During the set up of templates it also means defining the data for the import. Every property will be fulfilled with actual data, when the user publishes data from the Logica Application. So when creating a template the text that is marked for one property, does not matter, as it will be overwritten when the import takes place.

Table 1: The RDFa Buttons in the KiWi editor.

Icon	Title	Description
	Create/edit RDFa property	Marks the selected text to be the data of a property of the previously defined types of the page. Clicking on this icon opens a small dialog where the user can choose from the available properties. E.g., “John Doe” should be marked as <code>Employee.hasName()</code> .
	Create/edit RDFa link	Marks the selected text to be a link to another resource. The relations of a resource to another should be defined like this, to avoid the same data in multiple pages. It also helps to distinguish better between the different resources for the user. Clicking this icon opens a small dialog where the user chooses the type of the other resource first and then enters the name of the page. E.g., an employee is part of an organizational unit. Thus the employee’s details are on one page, the organizational unit’s on another, but both are linked together.
	Create component	Every page is of a certain type, these can link to other types/resources. In case the other resource should be presented on the same page, it can be encapsulated inside a component. Clicking this icon opens a small dialog where the user should choose one of the relations available. This defines the type of the component. Once a component is created it works like the standard text. E.g., putting the details of an organizational unit onto the page of an employee, defined through the relation <code>Employee.inOrganizationalUnit()</code> .
	Create/edit component list	If there is more than one resource of a relation to be put on the page, a component list is the obvious choice. It acts like a container for components. E.g., putting the employees that are in an organizational unit on the page of the latter one, each employee resource is one component in the list.



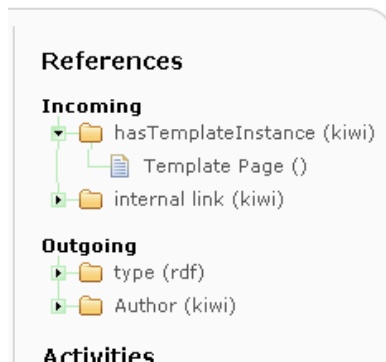


Figure 13: References from the template instance view

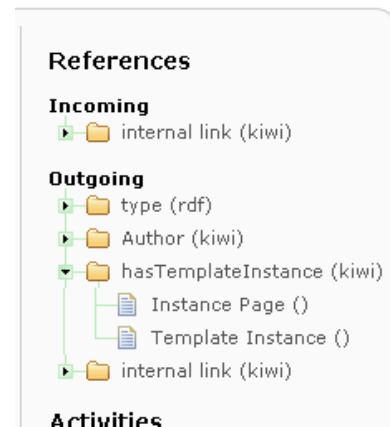


Figure 14: References from the template page view

Templates and their instances are related in KiWi. The connection is displayed in the references section, on the right of the text content on any KiWi page. Figure 13 shows the incoming relation of the template (title: “Template Page”) on an instantiation page. Figure 14 shows the outgoing relation of a template page to its instantiation pages (here titles “Instance Page” and “Template Instance”). These references are maintained and displayed by the KiWi system automatically to ease the navigation.

### 5.2.2. Data Exchange Application

The DxA is, unlike the two other systems, a desktop application. After its start up, it connects to the KiWi System to retrieve the templates. Only data fields defined in templates can be published. Then it accesses the database of the Logica Application, to show the specific data that can be published. Note that the data presented in the DxA is always coming from the Logica Application.

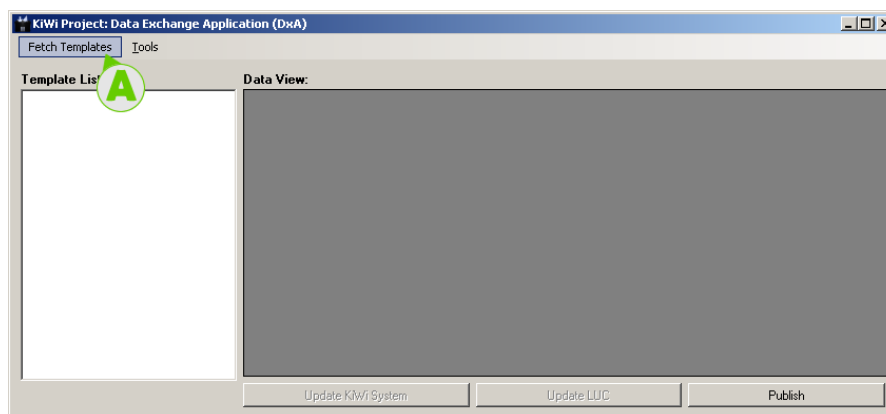


Figure 15: The DxA right after start-up.

After the start-up, the DxA has to ask the KiWi system for included templates (so: pages of type `kiwi:Template`) first. The user can trigger that by clicking on “Fetch Templates” in the menu (A) (Figure 15).

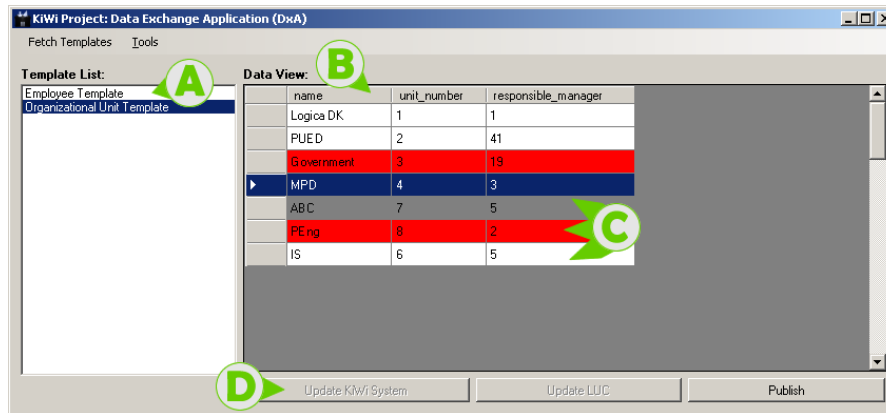


Figure 16: The DxA data overview.

When fetching the templates is finished, a list of templates in the KiWi system is displayed (A in Figure 16). After selecting one of them, the “Data View” (B) is updated and displays now a table. This table organizes the fields that are included in the templates inside the KiWi system and the relating data inside the Logica application. Each row represents a possible instance of the template. The background colors of the data rows in this table (C) indicate already published data, to give a better overview for the user. The meaning of the different colors is described in Table 2.

Table 2: The color meanings in the DxA’s data view.

Background Color	Meaning	Buttons
<input type="checkbox"/> white	The data in this row has <i>not</i> been published before.	only “Publish”
<input type="checkbox"/> grey	The data in this row has been published before and the data in both applications is the same (KiWi and Logica are in sync).	none
<input checked="" type="checkbox"/> red	The data in this row has been published before and the data differs from one application to the other (KiWi and Logica are out of sync).	“Update KiWi System” and “Update LUC”

As soon as the user selects a row (selected rows are always marked in blue) the buttons with the possible actions (D; “Publish”, “Update KiWi System” and “Update LUC” – these correspond directly to the use cases defined in Section 4) are activated or deactivated, according to the choice. “Publish” only works if the data has no representative in the KiWi system (and thus is not published, yet). If the selected row is represented in the KiWi system already and the data in both systems are *not* equal, the user can choose between updating the data inside the KiWi system or the Logica application. It is important to know, that the DxA does not provide further information here. It shows the data inside the Logica application and tells that the corresponding data in the KiWi system is not the same. Due to its loose coupling, the DxA has no information about changes in the applications or which data it published. It relies on a responsible user.

A small pop-up window acknowledges the successful process of the publishing or updating. After the work with it can the user close the DxA. Users do not have to save any settings or maintain the connection.

### 5.2.3. Logica Application

The Logica Application is a project management system with a complex data model. Here we will give an overview about how to handle the application. This means how to insert, update or delete data. From this perspective the application is rather simple to handle. Use the top-navigation (A in Figure 17) to jump to the entity page that you want.

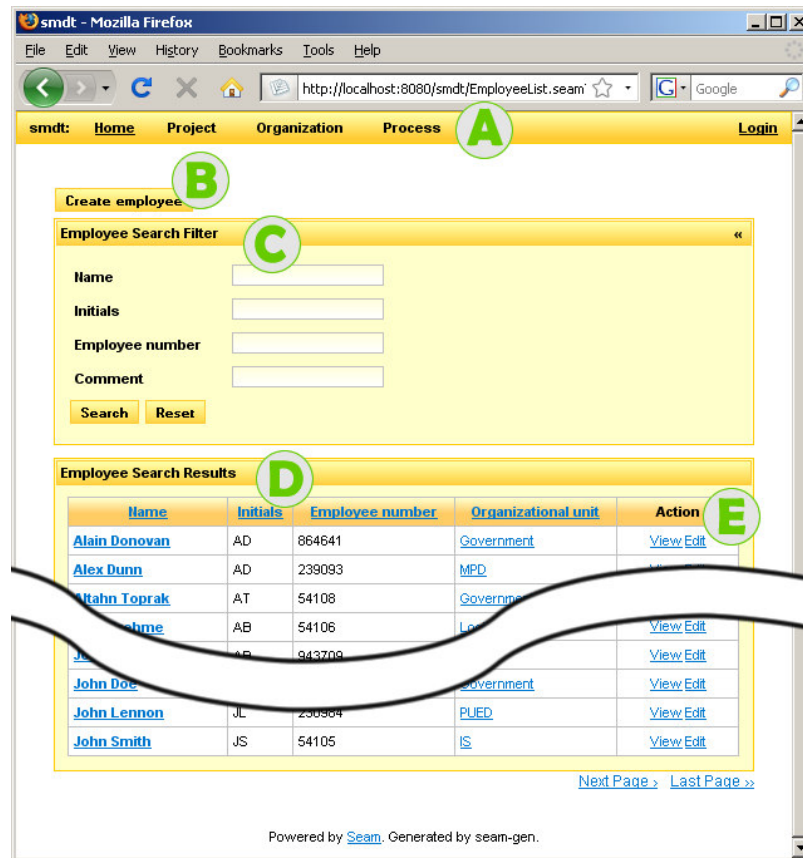


Figure 17: An entity list screen in the Logica Application.

Each entity page is built similarly, like in Figure 17. There is a button on top to create a new entity (B), clicking it leads to the insert/edit screen, which we describe afterwards. Below the create button is the search mask (C), it helps filtering for certain entities. If filters are set, the result set is displayed afterwards. Otherwise all entities are displayed in a table (D). Note that related entities are links (blue color, underlined and bold) in the table. To view an entry, you can either click the link or the field “View” in the last column (E).

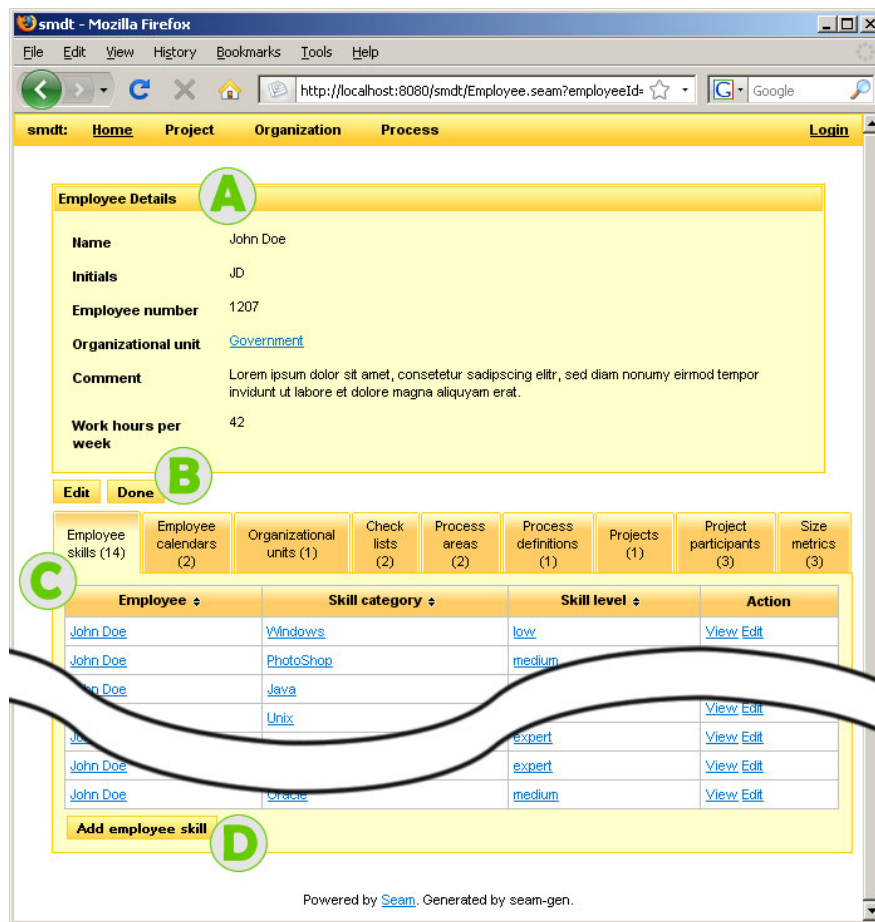


Figure 18: An entity's details page.

The details page of one entity is shown in Figure 18. It is basically organized into two parts, attributes and relations. The first box (A) shows all attributes. If this entity relates to another one, no matter of what type you can see a link to that (here: The Organizational Unit of which the Employee is a member). Note that only outgoing relations are presented like this. Below the details there is an “Edit” button which allows to edit the displayed details and a “Done” button which leads back to the list of entities. The second part organizes the entities that link to this one, so incoming references (C). The different types are organized in tabs. New entities of these can be created directly by clicking on the button at the bottom (D).

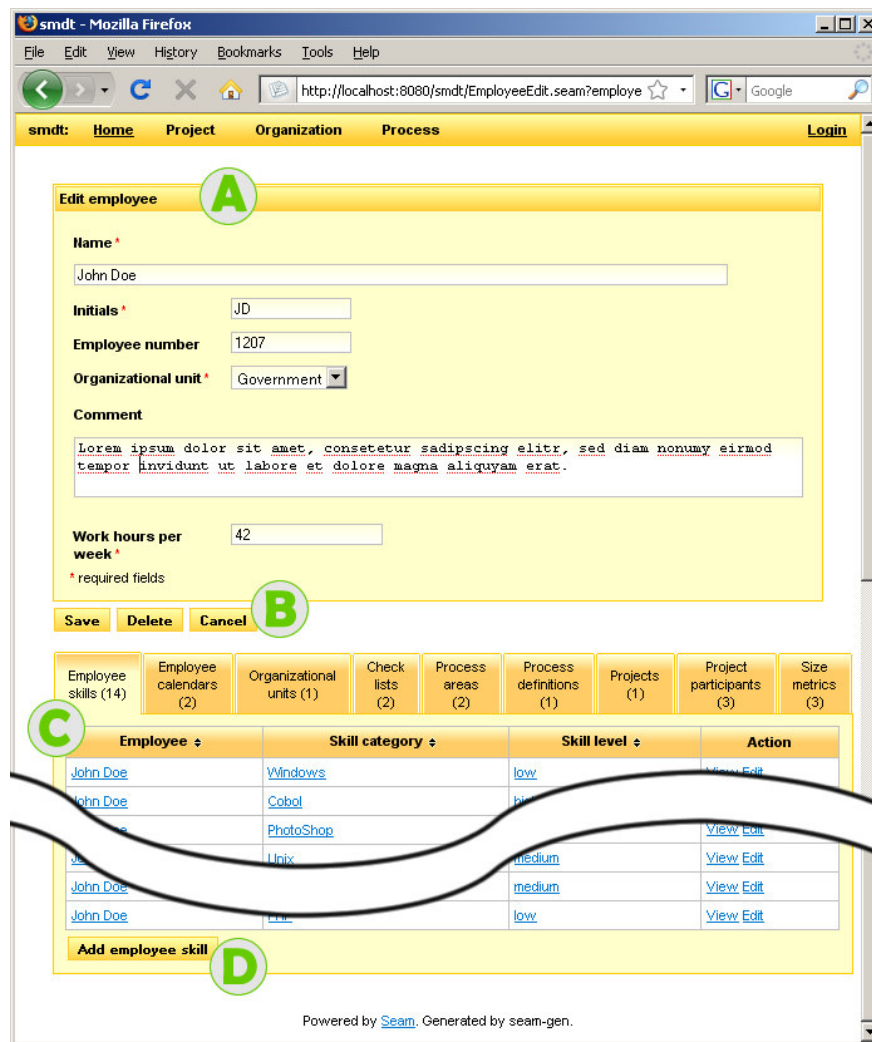


Figure 19: The edit page for an entity.

The only view missing for the Logica Application is the edit-screen, presented in Figure 19. It is similar structured as the details page; the main difference is that the attributes are filled in a form now and thus changeable by the user (A). Note that the reference is maintained in a list of options. After editing the user can choose from three buttons (B); “Save” to persist the changes, “Delete” to delete the entity from the database and “Cancel” to discard the changes and return to the details view. Below is the box with the entities that reference to this one (C) and the button to add additional ones (D) like in the details page.

Creating a new entity both works and looks exactly like editing one. The only difference is that the form fields are empty. Note if a delete button is clicked, the system deletes the chosen entity directly. The user is lead back to the entity list then.

## 6. Logica Use Case Application Components

This section gives an overview about how the different parts and functionalities used to realize the project knowledge management use case. We describe their background, how they are working and what they are used for.

### 6.1. Knowledge Model

The shared knowledge model between the Logica application and the KiWi system is what makes

the *circle of knowledge* possible. The DxA's task is it to map the data from one systems knowledge model to those from the other. It is possible to find a representative for every field of the Logica application's database in the project management ontology. Such completeness is mandatory to be able to publish any data. In an earlier stage of the KiWi Project, a first Version of this ontology was created (Dolog, et al., 2009). However, during the development for the use case it changed quite dramatically. For simplicity reasons we decided to base both, the ontology as well as the Logica application on the same data model. This data model was created at Logica, using relational data technologies.

While working for the original version (as delivered) of the knowledge model, we created an ontology that covers the major objectives of project management. But to provide full data integration, we needed a more complete one, where the data from the Logica application can completely be mapped to. The Logica application's relational database contains more than 60 tables. For obvious reasons, we were not able to create a fully qualified ontology based on that in the short amount of time. So we took the originally delivered project management ontology as a base and extended it. We created a class with properties from every entity and its fields in the relational database and added all of them to the ontology. If the concept existed already, then we edited it.

The resulting ontology was rather a list of concepts and properties. The property `hasName` for example was defined for 40 domains. During the following refactoring, we tried to pick some pieces of poor design and fixed them. E.g., we introduced a `logica:DescriptionClass` with properties like `hasName` or `hasDescription` and made it a super class for other classes. As an example take a look at the `Employee` class, which did not exist in the original delivered ontology.

```
<owl:Class rdf:ID="Employee">
  <rdfs:subClassOf rdf:resource="#ProcessDefinitionClass"/>
  <rdfs:subClassOf rdf:resource="#ProjectParticipantClass"/>
  <rdfs:subClassOf rdf:resource="#CommentClass"/>
  <rdfs:subClassOf rdf:resource="#OriginalIdentifierClass"/>
  <rdfs:subClassOf rdf:resource="#DescriptionClass"/>
  <rdfs:subClassOf rdf:resource="#ProcessAreaDefinitionClass"/>
  <rdfs:label rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >Employee</rdfs:label>
</owl:Class>
```

Figure 20 shows a graph of the `Employee` class. For readability reasons did not include property links here, but the properties are included.

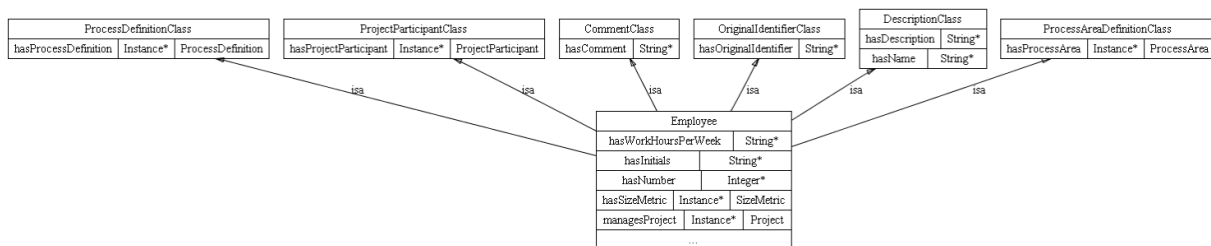


Figure 20: Graph of the `Employee` class.

We leave the provided information and refer to the delivered document (Dolog, et al., 2009), where the main concepts are described. Apart from the needed concepts we created a set of parental classes, which can be inherited from:

- **CommentClass**  
Defines a property to add a comment
- **DescriptionClass**  
Containing properties like `hasName` to describe a entity
- **OwnerClass**  
Defining the relation to an owner resource
- **PeriodClass**  
Defining a date and time period, with start and end point.
- **ProcessAreaDefinitionClass**  
Defining the relation to a resource of type `ProcessAreaDefinition`
- **ProcessDefinitionClass**  
Defining the relation to a resource of type `ProcessDefinition`
- **ProjectClass**  
Defining the relation to a resource of type `Project`
- **ProjectPlannedClass**  
Defining the relation to a resource of type `ProjectPlanned`
- **RiskyClass**  
Defining the relation to a resource of type `Risk`
- **StakeholderClass**  
Defining the stakeholder for a resource
- **TaskClass**  
Defining the relation to a resource of type `Task`
- **TaskDefinitionClass**  
Defining the relation to a resource of type `TaskDefinition`
- **WorkProductClass**  
Defining the relation to a resource of type `WorkProduct`

The ontology was mainly build with *Protégé*<sup>5</sup>, an ontology editor tool which provides facilities for creating classes as entities and establishing relationship between them though semantic properties. The created version contains a variety of different concepts and can likely be applied in multiple different environments (thus: with different ERP systems) as well. However, we respect the flexibility of semantic knowledge models, the mapping of the DxA from the ontology to the relational data schema can be updated in an XML file without the need for recompilation.

## 6.2. Enabling Technologies

For a successful running system, the project management use case makes use of all enabling technologies from the KiWi project.

### Information Extraction

The functionality to relate text parts to the knowledge model is part of the information extraction development. It realized through RDFa. This feature allows publishing data without the need to change any text content. The import exchanges data directly which is automatically included into the displayed text on the wiki page. Details on the implementation can be found in Section 6.3 where we discuss the templates.

---

<sup>5</sup> <http://protege.stanford.edu/>

## Reasoning

Every page that is used as a template has to be assigned to a type from the knowledge model. Thus the system knows what data to import or where to link to. It is quite likely that the type inherits attributes from parent types. In our case for example can a page describe employees and is as such of type `logica:Employee`. As described above is this a subclass of `logica:DescriptionClass` that contains attributes like `hasName`. Of course these are needed as well, but the user should not bother to add parental classes. In fact users are most likely not even aware of the fact that parental types contain desired properties. So, the KiWi reasoner runs as a background process and takes care of this automatically. After the user added a type (and saved her changes), the reasoner adds parental types explicitly to each content item, so that for example their properties are usable in template definitions.

## Personalization

Data import provides the ability to have a valuable personalization even on early stages in a project. The personalization unit accesses the published data and can start recommendation services from that knowledge base. An example, the project manager enters team members into his project management application, the process to be followed and assigns the resulting activities. These are the first steps in a project. Following the manager publishes all that data to the KiWi system. No the recommendation services have access to a knowledge base that contains people data, their assignments, their colleagues and the project's process.

The use of personalization benefits the work of the project's team inside the KiWi system. It is one of the reasons why we want to import data into it and thus move the project knowledge sharing work into a dedicated system.

## 6.3. Templates

The idea of having templates inside the KiWi System is, to make semi-structured data readable for users. Basically this consists of text blocks with placeholders that are substituted during the instantiation process by real data.

To do so we distinguish two different kinds of templates: Type-Templates and Data-Templates. We will describe both of them in detail later. First we want to focus on the template mechanism in general, as both have that in common.

### 6.3.1. Template Mechanism

Templates are pages in the KiWi System. Each one describes a certain set of information. The data however, is not real data yet. We call these placeholders. They are visible in the text and in the RDF data of a page. In the text, because users that view and/or edit the template have to know where a specific data field should be placed. It is in the knowledge base as well, because the text content is wired to the RDF data.

Let's look at a simple example. The page "Project-Resource Template" looks like this for a user:

```
From PROJECT_RESOURCE.TO_BE_USED_FROM to PROJECT_RESOURCE.TO_BE_USED_TO the resource
PROJECT_RESOURCE.NAME, PROJECT_RESOURCE.NUMBER_OF_RESSOURCE is used.
```

The relating RDF looks like this:

```
prtempl pr:usedFrom "PROJECT_RESOURCE.TO_BE_USED_FROM" .
prtempl pr:usedTo "PROJECT_RESOURCE.TO_BE_USED_TO" .
prtempl pr:hasName "PROJECT_RESOURCE.NAME" .
prtempl pr:hasNumber "PROJECT_RESOURCE.NUMBER_OF_RESSOURCE" .
```



The placeholders are highlighted in green. The template page forms a sentence that is easy to understand by users, if the placeholders are substituted here. An appearing problem is, if I substitute the placeholders in the page, I would have to do that in the RDF data as well. This leads to maintenance overhead. To get rid of this, we use RDFa for the template.

RDFa allows annotating parts of a textual content, which will be saved as RDF triples in the triple store. By editing these annotated parts, the triples in the triple store will be updated. If the triples change, the annotated text parts will also be updated. The underlying code for the page described above would look like this, if it was created with RDFa:

```
From <span property="pr:usedFrom">PROJECT_RESOURCE.TO_BE_USED_FROM</span> to  
<span property="pr:usedTo">PROJECT_RESOURCE.TO_BE_USED_TO</span> the resource  
<span property="pr:hasName">PROJECT_RESOURCE.NAME</span>,  
<span property="pr:hasNumber">PROJECT_RESOURCE.NUMBER_OF_RESSOURCE</span> is used.
```

The RDFa tags are highlighted orange; the placeholders are still there (green). They now are annotated, so the system knows what property to fill with the text.

This defines the main functionality of templates. Further features are special to the kind of template.

### 6.3.2. Type-Templates

The type-template represents data from one specific class only. A page is directly related to a type. The information in here is directly related to the class it belongs to. Relations are marked as such and work as links to other pages.

This kind of template represents a complete view on the data. Whenever users want to look at a resource, the system provides the data by using these templates. The pages for this are not explicitly created, but set up the moment a new resource is introduced.

### Implementation Details

The templating system will produce a text content from a given RDF graph and a single URI of an entity, based on some template definitions. The templates are linked to the types of the entities, so the KiWi knows which template to use based on the type of the entity (as described in the RDF graph).

All the information on an instance shall be represented using these mechanisms:

1. *Datatype Properties* - Correspond to triples with literal objects. The content of the property can be edited directly on the page and this modification will directly modify the literal value. PROJECT\_PLAN.VERSION is an example of a property.
2. *Object Properties (aka links)* - are semantic links, that is links with an associated predicate which defines a relationship between the page that the link is contained in and the page that the link is linking to. A relation may be modified only by selecting some other page to link to. It does not allow arbitrary content to be put on the place of the relation. There may also be a type safety mechanism to allow linking only to pages with a proper type, as defined in the template.

In the current implementation the links described here distinguished from the usual KiWi links by the choice of the element name (<a/> for usual links, <span/> for the semantic links.) The reason is so that both the KiWi link plug-in and the RDFa link plug-in can co-exist in the current system.

3. *Nested Content Items* - Allow structuring of the page, e.g. to allow multiple project resources on one project plan page. The components are thus useful for adding additional information how other resources relate to the current page, when a simple relation (link) is not enough. For example, we can create a component with a link to a project resource and add properties TO\_BE\_USED\_FROM and TO\_BE\_USED\_TO, which can be seen as properties of the relation between the project plan and the resource.

The templates for the project plan type (`logica:ProjectPlan`) could, for example, look like this:

```
<span property="logica:hasName"/> is a project plan of the
<span rel="logica:belongsTo" typeof="logica:Project"/> project. It is currently in
version <span property="logica:hasVersion"/> and it is following the
<span rel="logica:hasLifeCycleModel" typeof="logica:LifeCycleModel"/> model.
<div kiwi:iteratedinclude="logica:hasProjectResource"/>
```

The template for the project resource component type (`logica:ProjectResource`) could then be something like:

```
From <span property="logica:hasStartDate"/> to <span property="logica:hasEndDate"/> the
resource <span property="logica:hasName"/>,
<span property="logica:hasNumberOfResources"/> is used.
```

To instantiate this template (which is in fact two templates, where instances of the first will contain instances of the other as nested content items), one would only have to provide the structured RDF data:

```
@prefix rdf:      <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs:     <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl:    <http://www.w3.org/2002/07/owl#> .
@prefix foaf:     <http://xmlns.com/foaf/0.1/> .
@prefix logica:   <http://www.owl-ontologies.com/Logica.owl#> .
@prefix:          <http://example.com/logica/> .

:projX a logica:Project .
:projX rdfs:label "ProjectX" .
:projX logica:hasName "ProjectX" .

:lcml a logica:LifeCycleModel .
:lcml rdfs:label "Waterfall" .

:plan1 a logica:ProjectPlan .
:plan1 logica:hasName "Z123" .
:plan1 rdfs:label "Z123" .
:plan1 logica:hasVersion "1.0" .
:plan1 logica:hasLifeCycleModel :lcml .
:plan1 logica:hasProjectResource :x1 .
:plan1 logica:hasProjectResource :x2 .
:plan1 logica:belongsTo :projX .

:x1 a logica:ProjectResource .
:x1 rdfs:label "Office22" .
:x1 logica:hasName "Office 22" .
:x1 logica:hasStartDate "2010-01-01" .
:x1 logica:hasEndDate "2010-05-01" .
:x1 logica:hasNumberOfResources "1" .

:x2 a logica:ProjectResource .
:x2 rdfs:label "CoffeMachine" .
:x2 logica:hasName "Coffee Machine" .
:x2 logica:hasStartDate "2010-02-01" .
```

```
:x2    logica:hasEndDate "2010-05-05" .
:x2    logica:hasNumberOfResources "2" .
```

This and the type definitions would be enough to automatically construct the instance (plus the title of the pages linked to which are already stored in the KiWi and are used as default labels for links). The result would look like this (the RDFa output):

```
<span property="logica:hasName">Z123</span> is a project plan of the
<span rel="logica:belongsTo" typeof="logica:Project"
  resource="example:ProjectX">ProjectX</span>
project. It is currently in version
<span property="logica:hasVersion">1.0</span> and it is following the
<span rel="logica:hasLifeCycleModel" typeof="logica:LifeCycleModel"
  resource="example:lcm1">Waterfall</span> model.
<div rel="logica:hasProjectResource">
  <div about="example:x1">
    From <span property="logica:hasStartDate">2010-01-01</span> to
    <span property="logica:hasEndDate">2010-05-01</span> the resource
    <span property="logica:hasName">Office 22</span>,
    <span property="logica:hasNumberOfResources">1</span> is used.
  </div>
  ...
</div>
```

Internally, the component would be stored as a separate content item, so the KiWi iterated include would be expanded and stored internally like this:

```
<div kiwi:iteratedinclude="logica:hasProjectResource">
  <div kiwi:component="example:x1"/>
  <div kiwi:component="example:x2"/>
</div>
```

The `kiwi:iteratedinclude` attribute needs to be preserved for the refresh operation, so it is possible to add other components to the page to where they belong without destroying the content.

### 6.3.3. Data-Templates

A data-template builds a data compilation from different resources. The page is, as with the type templates, directly related to a type, but the data on it can differ. This might be in two ways. First, the data represented on the page is not necessarily complete. A template can contain subsets of the properties the class describes. And secondly, other types can be included. Related data can be shown directly and not only through a link.

The main idea of this template kind is to be able to create pages and to define what data should be represented on them. Users can edit the content and decide what data is important enough to be viewable and what is not.

#### Implementation Details

To create a data-template, a new `ContentItem` of the RDF-type `kiwi:Template` must be created. Furthermore, the `ContentItem` must be typed with another RDF-type, for example `foaf:Person`, to be able to use RDFa to annotate parts of the text content. The text content contains free written wiki-text and may be annotated with RDFa as required by the author. This allows us to have highly flexible and yet structured information inside of a `ContentItem`.

In case more than one type is needed inside of a data-template, KiWi-includes becomes necessary.

Therefore, we define a `kiwi:iterableInclude` element with the attributes 'rel' and 'target'. The attribute 'target' specifies the resource of another data-template that contains information about another type; 'rel' is a RDFa specific attribute that defines the property-relation between two data-templates.

The `kiwi:iterableInclude` element allows us to define more than one template-instance compositions inside of another template instance. The listings below illustrate a code snippet of a template with iterable including.

```
<div about="http://kiwi-project.eu/kiwi/person/profile">
Hello, my name is <span property="foaf:firstName">FIRSTNAME</span>
<span property="foaf:surname">SURNAME</span>. I was born on the
<span property="foaf:birthday">dd.mm.yyyy</span> and I know the following people:
<kiwi:iterableInclude rel="foaf:knows"
target="http://kiwi-project.eu/kiwi/person/profile/friends" />
</div>
```

```
<div about="http://kiwi-project.eu/kiwi/person/profile/friends">
<span property="foaf:firstName">FIRSTNAME</span>
<span property="foaf:surname">SURNAME</span>.
</div>
```

During the template instantiating process, the `kiwi:iterableIncludes` will be translated into div elements that point to the composed template instances, e.g.

```
<div about="http://kiwi-project.eu/kiwi/person/profile/StephanieStroka">
Hello, my name is <span property="foaf:firstName">Stephanie</span>
<span property="foaf:surname">Stroka</span>. I was born on the
<span property="foaf:birthday">06.09.1987</span> and I know the following people:
<div rel="foaf:knows"
kiwi:component="http://kiwi-project.eu/kiwi/person/profile/friends/KarstenJahn" />
<div rel="foaf:knows"
kiwi:component="http://kiwi-project.eu/kiwi/person/profile/friends/RolfSint" />
</div>
```

The amount of div elements with `kiwi:component` attributes depends on the number of iterable includes that have been sent by the DxA. For the example above, the DxA sends the following data:

```
<rdf:RDF>
<rdf:Description
rdf:about="http://kiwi-project.eu/kiwi/person/profile/StephanieStroka">
<foaf:firstName>Stephanie</foaf:firstName>
<foaf:surname>Stroka</foaf:surname>
<foaf:birthday>06.09.1987</foaf:birthday>
<foaf:knows
rdf:resource="http://kiwi-project.eu/kiwi/person/profile/friends/KarstenJahn" />
<foaf:knows
rdf:resource="http://kiwi-project.eu/kiwi/person/profile/friends/RolfSint" />
</rdf:Description>
<rdf:Description
rdf:about="http://kiwi-project.eu/kiwi/person/profile/friends/KarstenJahn">
<foaf:firstName>Karsten</foaf:firstName>
<foaf:surname>Jahn</foaf:surname>
</rdf:Description>
```

```
<rdf:Description>
  rdf:about="http://kiwi-project.eu/kiwi/person/profile/friends/RolfSint">
    <foaf:firstName>Rolf</foaf:firstName>
    <foaf:surname>Sint</foaf:surname>
</rdf:Description>
</rdf:RDF>
```

The data is stored in the triple store. During the rendering phase of the template instance, the RDF triples are included into the text of the template instance. The `div` elements that point to other `kiwi:components` will also be rendered into the `ContentItem`.

## 6.4. GUI

The RDFa KiWi plugin in the editor supports the functionality to create and edit the templates and fill or modify the values in the template instances.

The editor supports both: editing the templates and editing the template instances. In fact, a template is treated as any other wiki page. The page becomes a template just because it is used as such.

Creating and editing templates is considered an advanced operation, because the knowledge of RDF and the particular domain ontology is required. On the other hand, editing the values of a template instance requires no special knowledge and is comparable to the experience of filling a form.

There are 4 types of elements embedded directly to the editor content.

1. A datatype property field. Is used to edit literal values. It corresponds to an RDF property. It may be useful for editing names, product properties, etc.
2. An object property field. Is similar to an ordinary HTML link, but it forces the user to select an entity of a specific type. It corresponds to an RDF relation with a specific predicate, which can be defined in the template.
3. A nested content item. Is useful for structuring related information about an entity to better fit the knowledge model. Since in KiWi, one content item can represent only one URI, a nested content item may be used in such cases, where it is necessary to have information about multiple entities displayed on a single page. A nested content item creates a sub-page inside the current page with its own content and own metadata. This nested component is related to the „containing“ page by a specific predicate.
4. A nested content item list. Provides a place where the templating mechanism should insert all the nested content items related to the current content item with a specific predicate. (Such as a list of all project assignments of a user). A nested content item list automatically updates the list of nested content items based on current knowledge base. This is useful in cases, where new information about some entity (such as new tasks for an employee) are imported during the update operation.

### Creating templates

To create a template, user starts with an empty page, adds the types that the page should work as a template for, and one additional type `kiwi:Template`, which identifies the page as a template. User can then start to specify all the metadata, using one of the four types of RDFa elements.

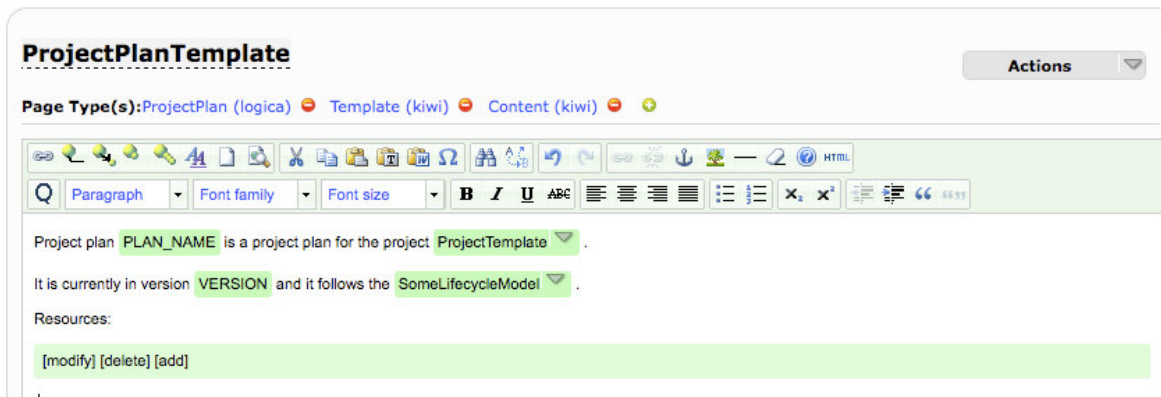


Figure 21: A Project Plan template with datatype properties (*hasName*, *hasVersion*), object properties (*belongsTo*, *hasLifecycleModel*) and one iteration (*hasProjectResource*).

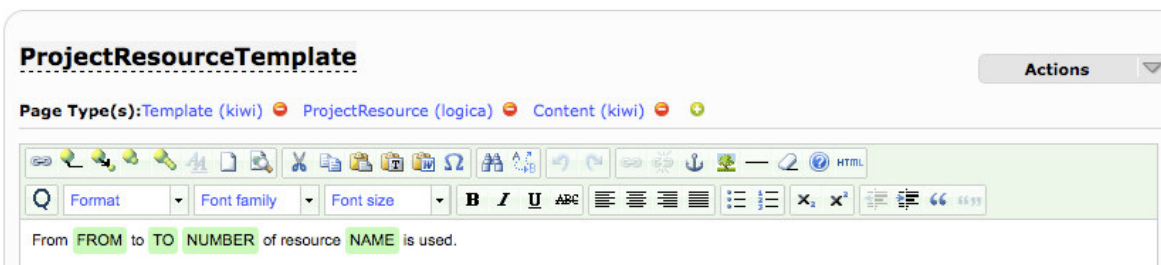


Figure 22: A Project resource template with datatype properties (*hasStartDate*, *hasEndDate*, *hasNumberOfResources*, *hasName*).

## Instantiating templates

User does not generally need to instantiate the templates manually. The instantiation is part of the importing service. The import service will import any RDF graph; find all the relevant templates for all the entities and instantiate the templates.

RDF data may be imported using the import tool (one has to check the “Use Templates” option to enable instantiation of templates. After the importing of the example data (described in n3 format above), a new content item for the project plan Z123 is generated:

```
Project plan Z123 is a project plan for the project ProjectX .
It is currently in version 1.0 and it follows the Waterfall .
Resources:
From 2010-02-01 to 2010-05-05 2 of resource Coffee Machine is used.
From 2010-01-01 to 2010-05-01 1 of resource Office 22 is used.
```

## 6.5. Web Services

The KiWi System contains two different kind of web services, service-oriented (through SOAP<sup>6</sup>) to publish data and RESTful<sup>7</sup> to ask for the containing data. Find a explanation of the two technologies and a comparison between them in (Pautasso, Zimmermann, & Leymann, 2008).

<sup>6</sup> SOAP: Simple Object Access Protocol; see: <http://www.w3.org/TR/soap/>

<sup>7</sup> RESTful Web Services: REpresentational State Transfer; HTTP based, see: <http://www.w3.org/TR/ws-arch/>

### 6.5.1. SOAP Web Service

The service-oriented interface contains basically one function for importing RDF data. It is part of the class `kiwi.api.webservice.DataAccessService`:

```
/**
 * The publishing method takes a rdf+xml formatted string and converts it into
 * triples, which are imported into KiWi. Additionally a new CI is created,
 * based on the given Template.
 * @param rdfdata The RDFdata that contains the data to be published
 * @param mimetype The mimetype, e.g., "application/rdf+xml"
 * @param templateURI The URI of the template in KiWi.
 * @param originURI The origin URI of the resource, external from KiWi.
 * @return The URL of the created CI or null if operation was not successful.
 */
@WebMethod
public String publishRDF(
    @WebParam String rdfData,
    @WebParam String mimetype,
    @WebParam String originURI,
    @WebParam String templateURI);
```

Based on the inserted values, the system creates a new content item for the external resource (field:originURI) with the RDF (rdfData) sent. Then it adds the text content and meta information of the template that is used for publishing (templateURI). A successful import returns a link to the new created content item, so that the user that handles the DxA can jump there directly.

The necessary WSDL<sup>8</sup> file can be found at this URL:

```
http://localhost:8080/KiWi/data_access?wsdl
```

### 6.5.2. RESTful Web Service

Relevant for the data import are two RESTful web services. One that retrieves a list of all templates (getTemplates) and one that retrieves the complete rdf to a given resource (getFields).

#### getTemplates

To get a list of all templates users can access a URL like this.

```
http://localhost:8080/KiWi/seam/resource/services/templateUpdateService/template
```

It returns an XML document like this:

```
<rdf:RDF xmlns:rdfs=http://www.w3.org/2000/01/rdf-schema#
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <Template xmlns="http://www.kiwi-project.eu/kiwi/core/"
    rdf:about="http://212.242.125.16:8080/KiWi/content/
      6f319311-6f3b-45ff-bed4-764cf8502f1a">
    <rdfs:label>Employee Template</rdfs:label>
    [...]
    <rdf:type rdf:resource="http://www.kiwi-project.eu/kiwi/core/Template"/>
    <rdf:type rdf:resource="http://www.owl-ontologies.com/Logica.owl#Employee"/>
    <hasWorkHoursPerWeek xmlns="http://www.owl-ontologies.com/Logica.owl#">
      HRS</hasWorkHoursPerWeek>
```

<sup>8</sup> WSDL: Web Service Definition Language; see: <http://www.w3.org/TR/wsdl>

```

<hasTextContent xmlns="http://www.kiwi-project.eu/kiwi/core/">
    [...]
</hasTextContent>
<hasInitials xmlns="http://www.owl-ontologies.com/Logica.owl#">
    INITS</hasInitials>

</Template>
</rdf:RDF>

```

Note that this represents just one template. A system that contains multiple templates returns multiple child nodes of RDF here, one for each template. Note also that the fields are not in any specific order and we left fields like internal id, language or title out here, to maintain readability. The main node contains the attribute `rdf:about` which represents the URI inside the KiWi System of the template.

### getFields

To retrieve the fields of any contentment item (including the templates) a second URL has to be accessed:

```

http://localhost:8080/KiWi/seam/resource/services/
templateUpdateService/template/{resource-URI}/fields

```

The URI of the resource has to be Base64 encoded. Only then the KiWi System can return an XML document like this:

```

<rdf:RDF xmlns:rdfs=http://www.w3.org/2000/01/rdf-schema#
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  <rdf:ContentItem rdf:about="http://212.242.125.16:8080/KiWi/
    content/6f319311-6f3b-45ff-bed4-764cf8502f1a">
      <hasWorkHoursPerWeek xmlns="http://www.owl-ontologies.com/Logica.owl#">
        HRS
      </hasWorkHoursPerWeek>
      <hasInitials xmlns="http://www.owl-ontologies.com/Logica.owl#">
        INITS
      </hasInitials>
    </rdf:ContentItem>
  </rdf:RDF>

```

Again the attribute `rdf:about` of the main node contains the URI of the resource inside KiWi and the fields do not follow any certain order.

The DxA uses the fields page to check whether data has been published already. It also uses this web service to check for changes of data. If the data changed, then an update might be needed.

## 6.6. Semantic Forms

Semantic forms bring the advantages of structured and form based systems by providing a controlled and easy way of filling, editing and displaying data to KiWi. Technically semantic forms are based on the RDFa templating mechanism described above.

To define a new semantic form, the creation of a template is required. A template may be any arbitrary wiki-page of the type `kiwi:template` and is used to define the structure of the form. Form fields are defined by placeholders, which are bound to the KiWi knowledge-model via RDFa. An instance of such a template is any arbitrary wiki page which has the type `kiwi:fromTemplate` and is connected via the property `kiwi:instancesTemplate` to the template. The template itself is connected via the property `kiwi:hasTemplateInstance` to the instance. Of course several



instances can be linked to one template.

Once an instance is created and connected to the template, all the RDFa placeholders defined in its corresponding template are included and replaced by inline-editable form fields.

Each KiWi ContentItem which has the type `kiwi:hasTemplate` will automatically include a form according to the form defined in its template.

Figure 23 illustrates the definition of a template by using placeholders with RDFa and Figure 24 illustrates its concrete instance.

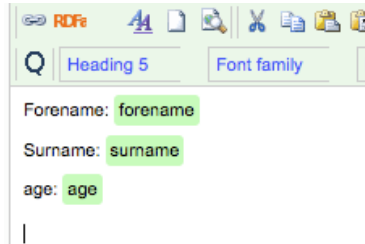
A screenshot of a web editor interface. At the top, there's a toolbar with icons for undo, redo, bold, italic, link, unlink, list, and table. Below the toolbar, there's a search bar and a dropdown menu showing 'Heading 5' and 'Font family'. The main content area contains three lines of text: 'Forename: forename', 'Surname: surname', and 'age: age'. Each placeholder is highlighted with a green background.

Figure 23: A template.

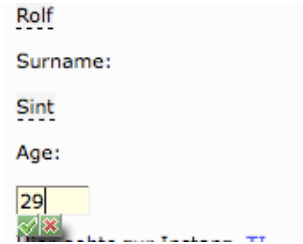
A screenshot of an inline edit form. It shows the instance of the template from Figure 23. The values are: 'Rolf' for Forename, 'Sint' for Surname, and '29' for age. The 'age' field is a text input box with the number '29' inside. Below the input box, there are two small icons: a green checkmark and a red X.

Figure 24: Inline edit form.

The changes and updates of the values within the form are automatically stored in the KiWi knowledge-model. Thus semantic forms provide a user-friendly way of creating new and modifying existing metadata to wiki-pages.

## 7. References

- Alavi, M., & Leidner, D. E. (2001). Review: Knowledge Management and Knowledge Management Systems: Conceptual Foundation and Research Issues. *MIS Quarterly* , 25 (1), 107-136.
- Dolog, P., Durao, F., Grolin, D., Karsten, J., Nielsen, P. A., Munk-Madsen, A., et al. (2009). *D6.3 Knowledge Model: Project Knowledge Management*. EU 7FP Project KiWi. Project Number: ICT-2007.4.2-211932; Document Number: ICT211932/AAU/D6.3/R/PU.
- Dolog, P., Grolin, D., Jahn, K., Munk-Madsen, A., Nielsen, P. A., & Pedersen, K. (2008). *Requirements Project Knowledge Management (KiWi Project Deliverable D5.4)*. EU 7FP Project KiWi. Project Number: ICT-2007.4.2-211932; Document Number: ICT211932/SRFG/D5.4/D/C/b1.
- Dolog, P., Krötzsch, M., Schaffert, S., & Vrandečić, D. (2009). Social Web and Knowledge Management. In *Weaving Services and People on the World Wide Web* (pp. 217-227). Springer Berlin Heidelberg.
- Dretske, F. I. (1981). *Knowledge and the Flow of Information*. Cambridge, MA: MIT Press.
- Machlup, F. (1980). *Knowledge: Its Creation, Distribution and Economic Significance, Volume I*. Princeton, NJ: Princeton University Press.
- Nielsen, P. A., & Dolog, P. (2008). *State-of-the-Art on Software Project Management Knowledge (KiWi Project Deliverable D5.3)*. EU 7FP Project KiWi. Project number ICT-2007.4.2-211932; Document number: ICT21193/SRFG/D5.3/D/PU/b1.
- Nonaka, I. (1994). A Dynamic Theory of Organizational Knowledge Creation. *Organization Science* , 5 (1), 14-37.
- Pautasso, C., Zimmermann, O., & Leymann, F. (2008). RESTful Web Services vs. "Big" Web Services: Making the Right Architectural Decision. *WWW '08: Proceeding of the 17th international conference on World Wide Web* (pp. 805-814 ). New York, NY, USA: ACM.
- Polanyi, M. (1966). *The Tacit Dimension*. London: Routledge & Kegan Paul.

Schaffert, S., Bry, F., Dolog, P., Eder, J., Gruenwald, S., Herwig, J., et al. (2008). *KiWi Vision (KiWi Project Deliverable D8.5)*. EU 7FP Project KiWi. Project number: ICT-2007.4.2-211932; Document number: ICT211932/SFRG/D8.5/D/PU/final.

Schaffert, S., Eder, J., Grünwald, S., Kurz, T., Radulescu, M., Sint, R., et al. (2009). KiWi - A Platform for Semantic Social Software. *Fourth Workshop on Semantic Wikis*. 464. CEUR-WS.

Wagner, C. (2004). Wiki: A Technology for conversational Knowledge Management and Group Collaboration. *Communications of the Association for Information Systems*, 13, 265-289.